基于存储改进的分区并行关联规则挖掘算法 *

王永贵，谢　南†，曲海成

(辽宁工程技术大学 软件学院，辽宁 葫芦岛 125105)

摘　要：基于关联规则在大数据挖掘领域正引起广泛关注，算法的重点及难点就是挖掘频繁集。针对现有算法存储结构简单、生成大量冗余的候选集、时间和空间复杂度高，挖掘效率不理想的情况。为了进一步提高关联规则算法挖掘频繁集的速度，优化算法的执行性能，提出基于内存结构改进的关联规则挖掘算法。算法基于 Spark 分布式框架，分区并行挖掘出频繁集，提出在挖掘过程中利用布隆过滤器进行项目存储，并对事务集和候选集进行精简化操作，进而达到优化挖掘频繁集的速度、节省计算资源的目的。算法在占用较少内存的条件下，相比于 YAFIM 和 MRApriori 算法，在挖掘频繁集效率上有明显地提升。算法不但能较好提升挖掘速度，降低了内存的压力，而且具有很好的可扩展性，使得算法可以应用到更大规模的数据集和集群，从而达到优化算法性能的目的。

关键词：关联规则；大数据；候选集；布隆过滤器；Spark

中图分类号：TP301.6　　doi: 10.3969/j.issn.1001-3695.2018.06.0396

Partitioned parallel association rules mining algorithm based on storage improvement

Wang Yonggui, Xie Nan†, Qu Haicheng

(*School of Software Liaoning Technical University*, *Huludao Liaoning* 125105, *China*)

Abstract: Association rules are attracting wide attention in the field of big data mining. The key and difficult point of the algorithm is to mine frequent sets. In order to further improve the speed of the association rules mining frequent sets and optimize the execution performance of the algorithm, an association rule mining algorithm based on improved memory structure is proposed. For the existing algorithm, the storage structure is simple, the candidate set with a large amount of redundancy is generated, the time and space complexity is high, and the mining efficiency is not ideal. The algorithm of this paper is based on the Spark distributed framework. The partitions are mined in parallel to extract frequent sets. It is proposed to use the Bloom filter to store the project in the mining process, and to simplify the operation of the transaction set and the candidate set, so as to optimize the speed of mining frequent sets. Save computing resources. Compared with the YAFIM algorithm and the MRApriori algorithm, the algorithm has a significant improvement in the efficiency of mining frequent sets under the condition of occupying less memory. The algorithm can not only improve the mining speed, reduce the memory pressure, but also has good scalability, so that the algorithm can be applied to larger data sets and clusters, so as to optimize the performance of the algorithm.

Key words: association rule; big data; bloom fileter; Spark

0　引言

关联规则起初是用来发掘购物篮中商品间的有趣关系，逐渐地成为数据挖掘的重要方法之一[1]。关联规则的目的是找出隐藏在大数据中的商品关系，重点是找出大数据集中的频繁项集，即那些频繁共同出现的商品集合。关联规则中最经典、影响最大的算法就是 Agrawal 等人提出的 Apriori 算法[2]，通过逐层搜索的迭代方式进行频繁项的挖掘，能够快速精准的挖掘出关联规则。随后出现了大量 Apriori 改进算法，但是都采用串行的方式，只有当提供的数据集很小时才会有较好的效果。

随着大数据时代的到来，集合中元素的增加，需要的存储空间越来越大，检索速度也越来越慢。各种单机的 Apriori 算法已经不能满足人们对时间和效率的需求。研究者们发现 Apriori 算法具有高并行性的可能性，因此，为了更高效地挖掘频繁集，引入了并行算法[3~5]。基于集群的关联规则算法能够提高处理事务数据集的效率，但是算法结构复杂，而且存在同步、数据复制等问题。研究发现基于 MapReduce 进行关联规则，挖掘频繁集效率更好，随后并行算法被 MapReduce 所取代，Apriori 算法基于

收稿日期：2018-06-05；**修回日期：**2018-07-26　**基金项目：**国家自然科学基金资助项目（61404069）；国家自然科学基金青年基金资助项目（41701479）

作者简介：王永贵（1967-），男，内蒙古宁城人，教授，硕士，大数据与数据仓库专家；谢南（1996　），男（通信作者），辽宁铁岭人，硕士，主要研究方向为数据挖掘（xienan0824@163.com）；曲海成（1981-），男，副教授，博士，主要研究方向为机器学习、数据挖掘.

MapReduce 的实现[6~8]被提出，与传统的 Apriori 算法相比显示出高性能增益。Apache Hadoop[9]是作为 MapReduce 模型的最佳平台之一。文献[10,11]基于 Hadoop 平台实现的 Apriori 算法，这些算法以 MapReduce 的方式对 Apriori 算法进行并行化处理，用 HDFS 存储数据集，在海量数据中发现频繁项集，实验表明该算法明显优于之前 Apriori 传统算法。但是基于 Hadoop 的 Apriori 算法的实现仍然存在一些限制。在 Hadoop 平台上，每次迭代后都会将结果存储到 HDFS 中，并从 HDFS 中获取输入以进行下一次迭代的过程导致性能降低。文献[12]中还指出 Hadoop 中 MapReduce 在迭代计算时任务启动和磁盘 I/O 开销过大使得执行效率降低，后来研究者们提出了 Spark 平台，可以很好地解决这些难题。

Spark [13]平台通过使用弹性分布式数据集架构 RDD 解决了这些问题，该架构在迭代结束时将结果存储在本地缓存中，并将它们提供给下一次迭代。对此提出基于 Spark 平台来改进 Apriori 算法，Spark 将数据基于内存计算，克服了上述 Hadoop 平台存在的问题，使其更加适用于在线、迭代和数据流算法。基于 Spark 平台，Zhang 等人[14]提出了一种分布式频繁项目集挖掘算法用于大数据分析，效果远远好于基于 Hadoop 平台实现的 Apriori 算法。上述算法基于线性表、链表、树等数据结构，这些数据结构，数据记录在结构中的相对位置是随机的，查找时进行大量关键字之间的比较，查找效率过于依赖查找过程中所进行的比较次数，随着数据量的增长，效率都将下降。

Qiu 等人[15]提出了 YAFIM 算法，基于 Spark 框架对大数据分区并行挖掘频繁集。该算法把候选集存储在哈希树中，被认为是目前较好的 Apriori 算法。实验结果发现 YAFIM 比基于 Hadoop 平台的算法要快很多倍。但是基于哈希树存储改进的 Apriori 算法需要先自连接生成候选集，再把候选集存储在哈希树中。当数据集大时，哈希函数很容易发生地址冲突，需要大量内存来完成每次迭代，第二次迭代出现最多候选集时内存占用尤为严重，严重影响算法效率。

在数据量很大的情况下，对数据结构中项目集的频繁增加和删除，容易导致数据结构退化为链表结构，当进行迭代过程，大量数据存储在内存中会对服务器造成巨大压力，挖掘频繁集性能必然将下降。因此，如何高效地判断一个项目是否在数据集中是提升算法执行速度的关键。布隆过滤器在底层只会使用几个 bit 来代表这个元素，并且对项目插入和查询时间都是常数，因此布隆过滤器在空间和时间方面有巨大的优势，方便 Apriori 算法的并行实现。基此提出把布隆过滤器应用到 Apriori 算法中，用布隆过滤器来改进 Apriori 算法存储结构，可以高效判断项目是否存在于数据集中。算法基于 Spark 分布式框架，对大数据集进行分区处理，随机分割成大小相似、非重叠的分区数据集，然后并行获取频繁集。本文算法可以明显节省内存空间的同时很好地提高数据处理的效率。

1　相关知识

1.1　Apriori 算法

Apriori 算法是一种挖掘布尔关联规则频繁模式集的算法[16]，首先遍历一次事务数据库，对所有一项集的计数，然后过滤到小于最小支持度的一项集，得到频繁一项集 L_1；然后进行迭代，由频繁 k-1 项集 L_{k-1} 进行连接剪枝生成候选 k 项集 C_k；最后过滤掉小于最小支持度的候选集得到频繁 k 项集 L_k，直到 L_k 为空时停止迭代，输出规则。算法主要思想就是通过定理 1 进行逐层搜索的迭代方式获取频繁集的过程。

定理 1　若一个模式不是频繁的，那么该模式的所有超集也都不是频繁模式.

现有的关联规则算法的数据结构简单，当处理大事务数据集时效率过低；所有事务始终位于文件系统或数据库，每次迭代都需要扫描事务数据库；Aprori 算法挖掘频繁集时，会生成大量项目集合，导致 I/O 开销过大且系统资源占用过多，因此时间性能和效率都很差。第二次迭代时，频繁一项集有 n 个，自连接会生成 2^n 个候选二项集合，候选集数量和占用计算资源是整个挖掘过程中最多的，过多导致算法第二次迭代挖掘频繁集过程耗时过久甚至无法继续运行。

1.2　Spark 平台

Spark 是一个围绕速度、易用性和复杂分析构建的大数据处理平台，提供全面、统一的框架。Spark 为交互式查询和迭代算法而开发，通过大数据查询的延迟计算，可以优化大数据处理流程中的处理步骤，支持内存式存储和高效的容错机制，将中间结果保存在内存而不是写入磁盘，使得 MapReduce 提升到更高层次。

在 Spark 平台下，将需要多次迭代的机器学习算法并行化，减少算法的时间复杂度和空间复杂度，在标准数据集上得到更高效的结果，这使得 Spark 平台适合实现 Apriori 算法。本文算法基于 Spark 大数据框架构建的分布式平台来改进 Apriori 算法，保证结果精确度的同时，解决了单机内存资源受限问题，并很好地提升了时间性能。在大数据背景下，算法能快速的、精准的进行关联规则挖掘。

1.3　布隆过滤器

布隆过滤器（Bloom filter）[17]是由一组很长的二进制向量和一系列随机散列函数组成的随机数据结构，这种数据结构用于判断一个元素是否在集合内。现构建一个长度为 M 位的数组布隆过滤器，且所有位都初始化 0、一组哈希函数 $H(x)=\{h_0(x),h_1(x),h_2(x)\dots\}$、事务 T={A,B,C}，事务中每个项目都通过 K 个不同的 hash 函数随机散列到数组的 K 个位置上，并将这些位置置为 1。

现要判断项目 w 是否在该事务 T 中，通过 hash 函数将事务 w 映射成位阵列中的点，只要看这些点是否都为 1 就知道元素是否在集合中。假设项目 w 通过三个哈希函数进行散列，发现出现有地址值为 0，可判断元素 w 不属于该集合，示例如图 1 所

示。

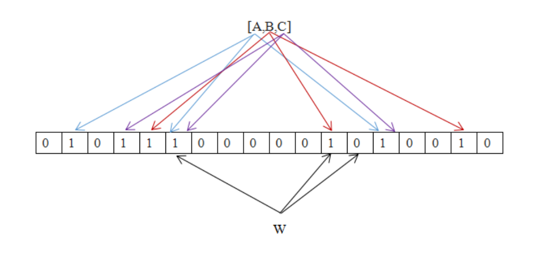


图 1　布隆过滤器示例图

布隆过滤器在判断一个项目是否属于该事务时，可能出现误判（false positive），即项目并不属于该事务但经过哈希散列后值都为 1。项目经映射后值都为 1 的概率 p（误判率）为

$$p=\left[1-\left(1-\frac{1}{M}\right)^{Kn}\right]^{K}\approx\left(1-e^{-Kn/M}\right)^{K} \tag{1}$$

化简为

$$p=e^{\left[K\ln(1-e^{-Kn/M})\right]} \tag{2}$$

令 g 为

$$g=K\ln(1-e^{-Kn/M}) \tag{3}$$

令 q 为

$$q=e^{-Kn/M} \tag{4}$$

可以将 g 改写为

$$g=-\frac{M}{n}\ln(q)\ln(1-q) \tag{5}$$

根据对称法则可以得到当 q＝0.5 时，即当 Kn/M=ln2 时，p 取得最小值。本文算法根据上述条件，自适应数据集，根据频繁一项集数目自适应选择哈希函数个数 K 与数组大小 M，进而减少不必要的散列操作，避免浪费时间空间且误判率最小。

2　算法

2.1　候选集的精简化

算法基于 Spark 框架，生成候选集的过程，首先压缩事务集，因为 C_k 一定是由 k 个项目组成，所以事务中项目个数小于 k 时，无法生成候选集 C_k，删除掉包含项目个数小于 k 的事务；然后压缩事务中的项目，根据定理 2 只保留存储在布隆过滤器中的频繁单项集，通过一个 map( )函数把这些频繁单项集组成所有可能的 k 项集；最后根据推论 1 压缩 k 项集，过滤掉出现次数不等于 k(k-1)/2 的 k 项集。本文算法不再进行冗余的连接步和剪切步来生成候选集，精简了事务集和候选集，从而减少每次迭代时需要扫描的事务、项目和候选集的数量，进而可以达到提升挖掘效率并节省计算资源的目的。

定理 2　一个频繁集的所有非空子集都是频繁集。

推论 1　一个频繁 k 项集 L_k，由 k 个 k-1 项集的构成，且出现了 k(k-1)/2 次。

证明　假设一个频繁三项集{A,B,C}可仅由两个频繁二项集{A,B}、{A,C}构成，这样{A,B,C}仅出现一次，即得出{B,C}不为频繁集，与定理 2 不符。所以{A,B,C}必由{A,B}、{A,C}、{B,C}构成即{A,B,C}出现三次，频繁 k 项集（k>3）依然如此，所以推论成立。

2.2　算法实现

本文算法基于 Spark 内部运行机制，读入 HDFS 的事务数据集加载到 Spark RDD 中，充分利用集群内存实现对弹性数据集 RDD 的并行计算，将数据集以 RDD 的形式存储，利用 textfile 算子扫描指定数据集并设置分片数，把数据集分为大小相似、非重叠的数据块，并分配到每个 work 节点进行处理。整个 Spark 编程框架均是基于 RDD 操作，结合 flatMap、map、reduceByKey、filter 等算子进行挖掘频繁集，使用 scala 语言进行编写。挖掘频繁集过程中把频繁集中的每个项目存储到布隆过滤器中。挖掘频繁集流程如图 2 所示。

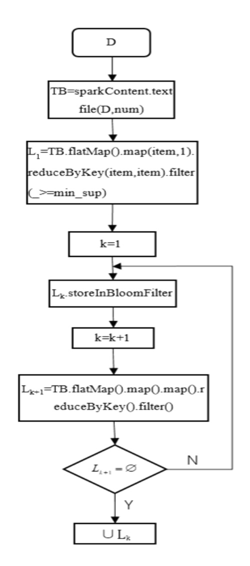


图 2　频繁集挖掘的流程

挖掘过程分为频繁一项集的获取和迭代生成频繁 k 项集俩个阶段。第一阶段寻找出所有的频繁一项集，读入 HDFS 中的待读取数据集。首先将 flatMap()函数应用于数据集中的每个分区数据集得到每个事务，再对于每个事务再应用 flatMap( )函数得到每一个项目；然后对每一项目应用 map( )函数生成<key,value>键值对，其中 key 表示事务中的每个项目，value 为这个事务的所有项目，本文将 value 设为 1，根据关键字通过 reduceByKey( )函数得到项目的计数<item,count>；最后通过 filter( )函数筛选出大于最小支持度阈值的项目，结果存储在 Spark RDD 并把频繁一项集存在布隆过滤器中。算法 1 为获取频繁一项集的算法，图 3 展示了频繁一项集生成的示例。

算法 1　获取频繁一项集

频繁一项集的获取

输入：数据集 D，最小支持度 min_sup

输出：频繁一项集 L1

```
1:  for each Transaction T∈D
2:     flatMap(T.split(“  ”))
3:     for each item i∈T
4:        yield(i, 1)
5:     end flatMap
6:     soreAtRDD1
7:  RDD2=RDD1.reduceByKey(_+_)
```

```
for each tuple t∈RDD2
    flatMap(t，count)
    if t.count>=min_sup
        yield(t，count)
    end flatMap()
    storeAtRDD3
```

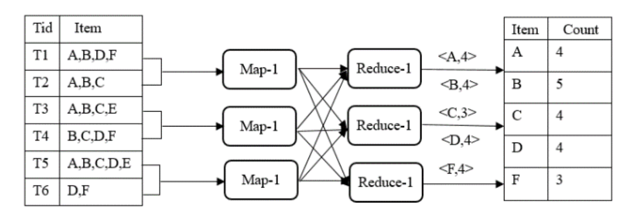


图 3　频繁一项集的生成的示例图

第二阶段开始进行迭代，由频繁 k-1 项集生成 k 项集。首先对 RDD 中待处理的数据集应用 flatMap( )函数，获取数据集中的每个事务，对每个事务应用 map( )函数进行修剪，删除掉项目个数小于 k+1 的事务，并且事务仅包含 Bloom 过滤器中存在的项目集；对修剪后的事务应用 map( )函数产生所有可能的 k 项集配对，生成<key,value>键值对，其中 key 表示事务中的 k 项候选模式，value 是事务中所有的 k 项候选模式集合，将它设为整数 1，并只保留出现次数为 k(k+1)/2 的 k 项候选集；根据关键字通过 reduceByKey( )函数组合所有键值对，并得到每个候选模式的计数，格式为<itemset,count>；最后通过 filter( )函数筛选出频率大于最小支持值的集合，把频繁 K+1 项集存储在 Spark RDD 和布隆过滤器中。当不再有频繁集生成时迭代结束，合并所有的频繁集合。算法 2 为获取频繁一项集的算法，图 4 展示了频繁二项集生成的示例。

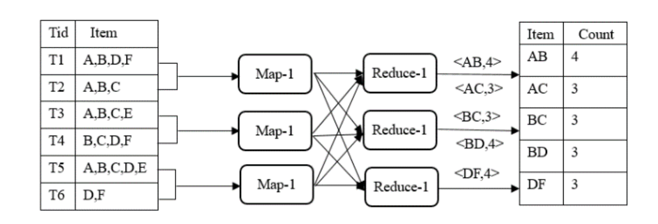


图 4　频繁二项集生成的示例图

算法 2　频繁 k 项集获取

频繁 k 项集的获取

输入：事务数据库 D，频繁 k-1 项集 Lk-1

输出：频繁 k 项集 Lk

```
Lk-1.storeInBloomFilter
for each Transaction T∈D
    flatMap(T.split(“  ”))
    item=intersect(T, Lk-1)
    C=map(item=>(itemSet,1))
    for each itemSet s∈C
        Ck=intersect(C，s.count= k(k-1)/2))
    end flatMap
    soreAtRDD1
for each Transaction T∈D
    flatMap(T.split(“  ”))
    CT=subset(Ck，T)
    for each canadiate c∈CT
        yield(c，1)
  end flatMap
  soreAtRDD1
  RDD2=RDD1.reduceByKey(_+_)
  for each itemSet∈RDD2
    flatMap(itemSet，count)
    if itemSet.count>=min_sup
        yield(itemSet，count)
    end flatMap()
    storeAtRDD3
```

2.3　算法的复杂度分析

第 K 次迭代，由频繁 K-1 项集生成频繁 K 项集，频繁 K-1 项集 L_{k-1} 的个数为 f，假设有 x 个事务，m 个 Mapper，每个事务中的平均项目个数为 g，n 为 k 次迭代生成的候选集个数，t_1 为在哈希树中搜索一个元素花费的时间，t_2 为在布隆过滤器中搜索一个元素花费的时间，t_3 为压缩过程所花时间，每个候选集占 b_1 个字节，每单个项目占 b_2 个字节。

第 K 次迭代，由频繁 K-1 项集生成频繁 K 项集，假设 f 为的频繁 K-1 项集个数，候选 K 项集 C_k 的个数为 n，有 x 个事务，m 个 Mapper，每个事务中的平均项目个数为 g，，t 为在哈希树中搜索一个元素花费的时间，b 为在布隆过滤器中搜索一个元素花费的时间。压缩时间 c

2.3.1 时间复杂度

生成候选集的时间（连接和剪切的总时间）T1，候选集存储到哈希树中的时间 T2，生成键值对的时间 T3，基于哈希树改进算法进行第 K 次迭代花费的时间为 O(Ta) 。

生成候选集的时间为

$$T1 = K*n^f \tag{6}$$

存储候选集到哈希树中的时间为

$$T2 = K*n^f \tag{7}$$

搜索和生成键值对的时间为：

$$T3 = \frac{x}{m}*t_1 g \tag{8}$$

哈希树改进算法的第 K 次迭代的时间复杂度为

$$Ta = O(2Kn^f + \frac{x}{m}*t_1 g) \tag{9}$$

存储频繁单项集在布隆过滤器中的时间 T4，剪切事务的时间 T5，生成键值对的时间 T6，利用布隆过滤器改进算法花费时间为 O(Tb)

频繁一项集存储在布隆过滤器中的时间：

$$T4 = t_2 \tag{10}$$

进行事务剪枝的时间：

$$T5=\frac{x}{m}*g+t_3 \tag{11}$$

在最糟糕的情况下生成键值对的时间：

$$T6=\frac{x}{m}*n^g \tag{12}$$

布隆过滤器改进算法第 K 次迭代时间复杂度：

$$Tb=O(t_2+t_3+\frac{x}{m}*(n^g+g)) \tag{13}$$

在大数据前提下，k（迭代次数），t_1（哈希树中搜索一个元素花费的时间），t_2（布隆过滤器中搜索一个元素花费的时间），t_3（事务和项目压缩的时间）都相对于其他值过小可以忽略不计，在迭代过程中 n 和 f 存在如下关系：

$$n=\frac{f(f-1)}{2} \tag{14}$$

在大数据环境下 n 特别大，因此 f 关于 n 的关系式可以化简得到

$$f=\sqrt{2n} \tag{15}$$

Ta-Tb 可化简为

$$2Kn^{\sqrt{2n}}+{}^{x}\!/_{m}\times t_1 g-{}^{x}\!/_{m}\times n^g \tag{16}$$

第二次迭代时候选集过大，$\sqrt{2n}>>g$ 即 Ta>>Tb。

由此可以得出，改进算法的时间复杂度远远低于 YAFIM 算法的时间复杂度。

2.3.2 空间复杂度

YAFIM 算法的空间复杂度是生成候选集和创建哈希树存储候选集所占内存的和。

存储候选集的空间复杂度为

$$O(c^f(k+1)\times b_1) \tag{17}$$

创建一个 hash 树空间复杂度为

$$O(k\times c^f\times b_2) \tag{18}$$

Apriori 算法总体空间复杂度

$$O(c^f[k(b_1+b_2)+b_1]) \tag{19}$$

本文算法在布隆过滤器中存储频繁集当中的单个项目，空间复杂度为 $O(b_2\times f)$。本文算法的空间复杂度远远小于 Apriori 算法的空间复杂度。

3　实验

3.1　实验数据

实验选择了六个特点不同的大数据集，如表 1 所示。

3.2　实验环境

实验的软件环境为 64 位 Ubantu 14.04 Linux 操作系统、JDK-1.8 Hadoop-2.6.0 Scala-2.12.1 Spark-1.6.1 Python-3.6.3 Anaconda-2.0；硬件环境为 Intel corei7-6500U 8 GB 内存 1T 硬盘。采用 8 台计算机搭建 Spark 集群运行环境，集群共 8 个节点，把其中一台计算机设为 master 节点，其他 7 台机器设为 slave 节点，每个节点 8 个 cores，64GB RAM。

表 1　数据集表

数据集	事务个数	项目个数
T1014D100K	100000	870
Retail	88163	16470
Musroom	8124	119
Kosarak	990002	41270
BMSWebView2	77512	3340
T25110D10K	4900	990

3.3　可扩展性

可扩展性[18]实际上是与并行算法以及并行计算机体系结构放在一起讨论的，某种算法在某个机器上的可扩展性反映该算法是否能有效利用不断增加的 cores[19]。本文研究可扩展性的目的就是要使算法可以利用更多的处理器，并且可以预测当某个算法移植到大规模处理机上的运行效果。

实验用增加逻辑核 core 的个数和数据集的复制倍数的方式，观察算法对不同数据集的执行时间，判断该算法的可执行性和可扩展性。本文选用了 Retail、Musroom、Kosarak、BMSWebView2 数据集进行实验，实验结果如图 5、6 所示。图 5 表示随着逻辑核个数的增加，执行时间几乎是呈现线性减少趋势；图 6 表示随着数据集复制倍数的增加，执行时间几乎是呈现线性增加的趋势，实验结果说明该算法是具有可执行性和可扩展性的。

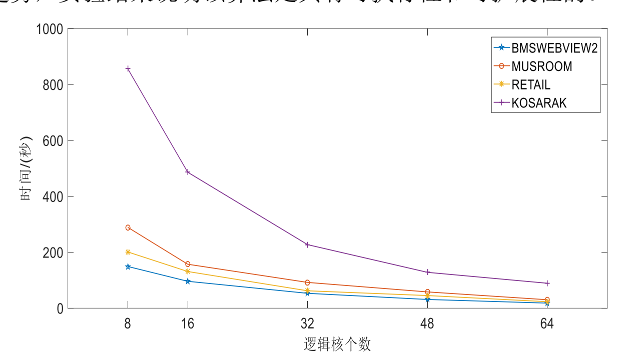


图 5　随着逻辑核个数的增多，算法执行时间的变化

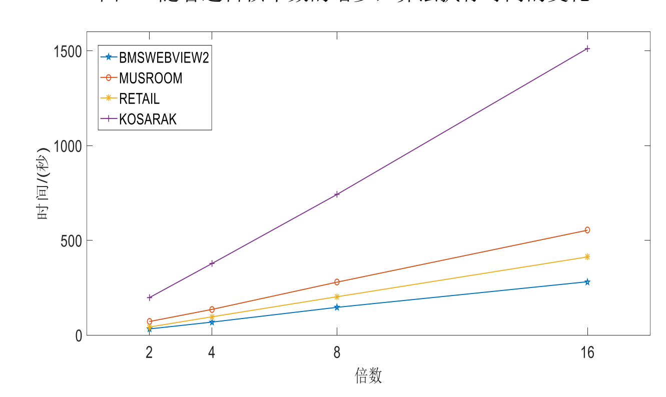


图 6　随着数据集复制倍数的增多，算法执行时间的变化

3.4　对比实验

对比实验选用基于 Hadoop 的经典算法 MRApriori 算法、基

于 Spark 上实现的 YFAIM 算法和本文算法，在同一数据集和最小支持度下进行比较。

在 Retail 数据集上，min_sup=0.15%的条件下三种算法的时间比较如图 7 所示。本文算法明显优于 YFAIM、MRApriori 算法，尤其在第二次迭代，生成频繁二项集时，本文算法在速度上比 YFAIM 算法提升了近 6 倍，比 MRApriori 算法提升了近 30 倍。

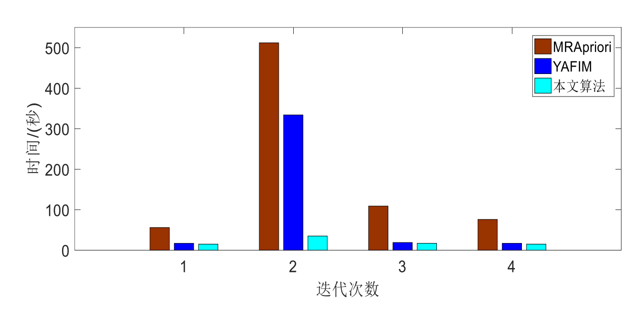


图 7　在 Retail 数据集下三个算法每次迭代的时间对比

在 T10I4D100K 数据集上，min_sup=0.15%条件下三种算法的比较时间如图 8 所示。本文算法明显优于 YFAIM、MRApriori 算法，尤其在第二次迭代生成频繁二项集时，本文算法在速度上比 YFAIM 算法提升了近三倍，比 MRApriori 算法提升了近 10 倍。

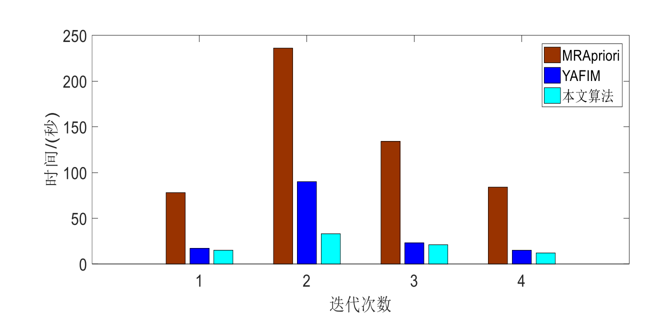


图 8　T10I4D100K 数据集下三个算法每次迭代的时间对比

在 T25I10D10K 数据集上，min_sup=0.10%条件下三种算法的比较时间如图 9 所示。本文算法明显优于 YFAIM、MRApriori 算法，尤其在第二次迭代生成频繁二项集时，本文算法在速度上比 YFAIM 算法提升了近三倍，比 MRApriori 算法提升了近 10 倍。

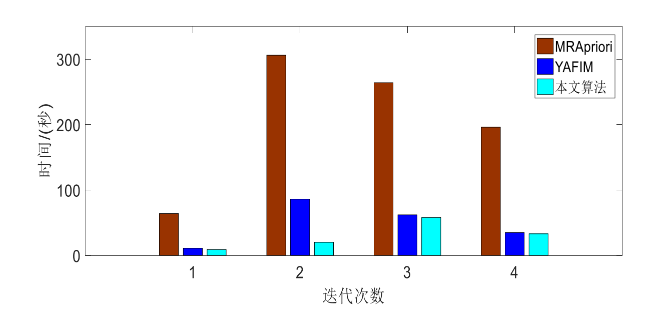


图 9　T25I10D10K 数据集下三个算法每次迭代的时间对比

4　结束语

本文提出基于存储改进的分区并行算法，围绕如何提升算法执行性能进行改进，充分利用布隆过滤器存储数据的时间和空间优势来满足挖掘过程中的存储、查询需要，避免大量数据存储在内存中会对服务器造成巨大压力。算法基于 Spark 框架，每次迭代，首先对事务进行压缩，并过滤掉未在布隆过滤器中出现的项目，然后基于频繁集中的单项组成候选集，压缩候选集，最后分区并行挖掘频繁集。实验结果表明，算法较好提升挖掘频繁项集效率尤其让第二次迭代变得高效，节省了资源空间同时具有很好的可扩展性，可以应用算法到更大规模的数据集和集群。

参考文献：

[1] 崔妍，包志强. 关联规则挖掘综述 [J]. 计算机应用研究, 2016, 33 (2): 330-334. (Chui Yan, Bao Zhiqiang. Summary of association rules mining [J]. Application Research of Computers, 2016, 33 (2): 330-334.)

[2] Mirjana M, Quintarelli E, Tanca L. Data mining for XML query-answering support [J]. IEEE Trans on Knowledge and Data Engineering, 2012, 24 (8): 1393-1407.

[3] Riondato M, DeBrabant J A, Fonseca R, *et al.* PARMA: a parallel randomized algorithm for approximate association rules mining in MapReduce [C]// Proc of the 21st ACM International Conference on Information and Knowledge Management. NewYork: ACM Press, 2012: 85-94.

[4] Wang Zuocheng, Xue Lixia. A fast algorithm for mining association rules in image [C]// Proc of the 5th IEEE International Conference on Software Engineering and Service Science. Piscataway, NJ: IEEE Press, 2014: 487-499.

[5] 张忠林，田苗凤，刘宗成. 大数据环境下关联规则并行分层挖掘算法研究 [J]. 计算机科学, 2016, 43 (1): 286-289. (Zhang Zhonglin, Tian Fengmiao, Liu Zongcheng. Research on parallel mining algorithm of association rules in big data environment [J]. Computer Science, 2016, 43 (1): 286-289.)

[6] Nguyen D, Vo B, Le B. Efficient strategies for parallel mining class association rules [J]. Expert Systems with Applications, 2014, 41 (10): 4716-4729.

[7] Lin M, Lee P, Hsueh S. Apriori-based frequent itemset mining algorithms on MapReduce [C]// Proc of the 6th International Conference on Ubiquitous Information Management and Communication. NewYork: ACM Press, 2012: 1-8.

[8] Lin Xueyan. Mr-apriori: association rules algorithm based on mapreduce [C]// Proc of the 5th IEEE International Conference on Software Engineering and Service Science. Piscataway, NJ: IEEE Press, 2014: 141-144.

[9] Yahya O, Hegazy O, Ezat E. An efficient implementation of A-Priori algorithm based on Hadoop-Mapreduce model [J]. International Sjournal of Reviews in Computing, 2012, 12 (7): 59-67.

[10] 林长方，吴扬扬，黄仲开，等. 基于 MapReduce 的 Apriori 算法并行化

[J]. 江南大学学报：自然科学版, 2014, 13 (4): 411-415. (Lin Zhangfang, Wu Yangyang, Huang Zhongkai, *et al.* Parallelization of Apriori algorithm based on MapReduce [J]. Journal of Jiangnan University: Natural Science, 2014, 13 (4): 411-415.)

[11] Saabith A L S, Sundararajan E, Abubakar A. Parallel implementation of apriori algorithms on the Hadoop-mapreduce platform-an evaluation of literature [J]. Journal of Theoretical and Applied Information Technology, 2016, 85 (3): 321-351.

[12] Zaharia M, Shixin R, Das T, *et al.* Apache Spark: a unified engine for big data processing [J]. Communications of the ACM, 2016, 59 (11): 56-65.

[13] Shanahan J G, Dai Laing. Large scale distributed data science using apache spark [C]// Proc of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. NewYork: ACM Press, 2015: 2323-2324.

[14] Zhang Feng, Liu Min, Gui Feng, *et al.* A distributed frequent itemset mining algorithm using Spark for Big Data analytics [J]. Cluster Computing, 2015, 18 (4): 1493-1501.

[15] Qiu Hongjian, Gu Rong, Yuan Chunfeng, *et al.* YAFIM: a parallel frequent itemset mining algorithm with Spark [C]// Proc of IEEE IPDPSW. Piscataway, NJ: IEEE Press, 2014: 1664-1671.

[16] 杨启昉，马广平. 关联规则挖掘 Apriori 算法的改进 [J]. 电子技术与软件工程, 2014, 28 (19): 199-200. (Yang Qifang, Ma Guangpin. Improvement of association rules mining apriori algorithm [J]. Electronic Technology and Software Engineering, 2014, 28 (19): 199-200.)

[17] Holley G, Wittler R, Stoye J. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage [J]. Algorithms for Molecular Biology Amb, 2016, 11 (1): 1-9.

[18] Li Li, Chou Wu, Zhou Wei, *et al.* Design patterns and extensibility of REST API for networking applications [J]. IEEE Trans on Network & Service Management, 2016, 13 (1): 154-167.